

PRIORITIZED EXPERIENCE DEEP DETERMINISTIC POLICY GRADIENT  
METHOD FOR DYNAMIC SYSTEMS

by

SERHAT EMRE CEBECİ

Submitted to the Graduate School of Engineering and Natural Sciences  
in partial fulfillment of  
the requirements for the degree of  
Master of Science

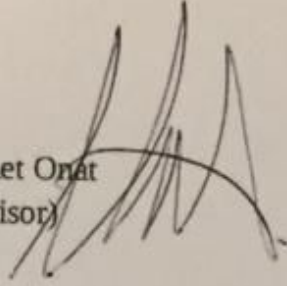
Sabancı University

July 2019

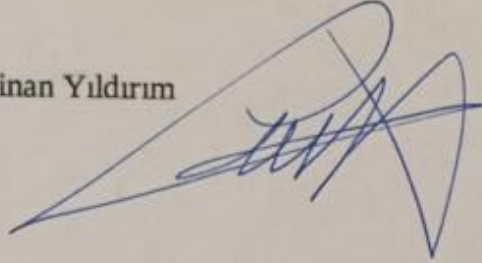
PRIORITIZED EXPERIENCE DEEP DETERMINISTIC POLICY GRADIENT  
METHOD FOR DYNAMIC SYSTEMS

APPROVED BY:

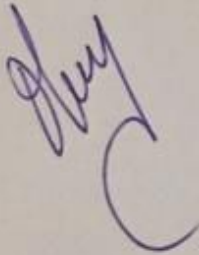
Assoc. Prof. Dr. Ahmet Onat  
(Thesis Supervisor)



Asst. Prof. Dr. Sinan Yıldırım



Prof. Dr. Ece Olcay Güneş



DATE OF APPROVAL: 01 / 07 / 2019

© Serhat Emre CEBECİ 2019

All Rights Reserved

# **PRIORITIZED EXPERIENCE DEEP DETERMINISTIC POLICY GRADIENT METHOD FOR DYNAMIC SYSTEMS**

SERHAT EMRE CEBECİ

Mechatronics Engineering, M.Sc. Thesis, 2019

**Thesis Advisor:** Assoc. Prof. Dr. Ahmet Onat

**Keywords:** deep reinforcement learning, neural networks, reinforcement learning, dynamic systems, deep learning

## **ABSTRACT**

In this thesis, the problem of learning to control a dynamic system through reinforcement learning is taken up. There are two important problems in learning to control dynamic systems under this framework: correlated sample space and curse of dimensionality: The first problem means that samples sequentially taken from the plant are correlated, and fail to provide a rich data set to learn from. The second problem means that plants with a large state dimension are untractable if states are quantized for the learning algorithm.

Recently, these problems have been attacked by state-of-the-art algorithm called Deep Deterministic Policy Gradient method (DDPG). In this thesis, we propose a new algorithm Prioritized Experience DDPG (PE-DDPG) that improves the sample efficiency of DDPG, through a Prioritized Experience Replay mechanism integrated into the original DDPG. It allows the agent experience some samples more frequently depending on their novelty. PE-DDPG algorithm is tested on OpenAI Gym's Inverted Pendulum task. The results of experiment show that the proposed algorithm can reduce training time and it has lower variance which implies more stable learning process.

# DİNAMİK SİSTEMLER İÇİN ÖNCELİKLİ DENEYİMLİ DERİN DETERMINİSTİK POLİTİKA GRADYAN YÖNTEMİ

SERHAT EMRE CEBECİ

Mekatronik Mühendisliği, Yüksek Lisans Tezi, 2019

**Tez Danışmanı:** Doç. Dr. Ahmet Onat

**Anahtar Kelimeler:** derin pekiştirmeli öğrenme, yapay sinir ağları, pekiştirmeli öğrenme, dinamik sistemle, derin öğrenme

## ÖZET

Bu çalışmada, pekiştirmeli öğrenme yoluyla dinamik sistemlerin kontrolünü öğrenme problemi ele alınmıştır. Dinamik sistemlerin kontrolünün öğrenmesi hususunda iki önemli problem vardır: ilintili örnek uzay ve çok boyutluluğun laneti: İlk problem, öğrenmek için kullanılan ardışık örneklerin, birbiriyle ilintili olmasından dolayı dinamik sistem kontrolünü öğrenmek için yeterli zengin veri setini sunamaması anlamına gelmektedir. İkinci problemse, büyük sayıda durum boyutuna sahip dinamik sistemler için durum uzayını niceliklerine ayırarak betimleme yaparak öğrenmek, öğrenilmesi gereken durum sayısını çok artıracak için, öğrenmenin imkansız veyahut çok zor hale gelmesi anlamına gelir.

Günümüzde, bu iki problem, güncel olan en iyi çalışma olan Derin Deterministik Politika Gradyan (DDPG) yoluyla çözülmeye çalışılmıştır. Bu çalışmada, Derin Deterministik Politika Gradyan yönteminin örnekleme yöntemini daha verimli bir yol olarak, Öncelikli Deneyimli Derin Deterministik Politika Gradyanı yöntemi öne sürülmüştür. Bu yöntem Derin Deterministik Politika Gradyanı yöntemine, Öncelikli Deneyim Tekrarı (Prioritized Experience Replay) yöntemindeki örnekleme yönteminin entegrasyonu olarak düşünülebilir. Bu yöntem ile, öğrenmenin her deneyimden eşit derece olmasının yerine, hatalı olan deneyimleri tekrar tekrar örnekleyerek, daha verimli öğrenmenin sağlanması amaçlanmıştır. Öncelikli Deneyimli Derin Deterministik Politika Gradyanı (PE-DDPG) yöntemi öne sürülmüş olup, bu yöntem OpenAI Gym aracındaki Ters Sarkaç problemi üzerinde test edilmiştir. Sonuçlar göstermektedir ki, önerilen yöntem öğrenme zamanını kısaltmış ve öğrenme sırasındaki varyansı da düşürerek daha kararlı bir öğrenme süreci sağlamıştır.

*Dedicated to my parents for their continued support throughout  
my life...*

# Contents

<b>Declaration of Authorship</b>	<b>i</b>
<b>Abstract</b>	<b>ii</b>
<b>List of Figures</b>	<b>v</b>
<b>List of Tables</b>	<b>vi</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Literature Review . . . . .	1
<b>2 The Reinforcement Learning Problem</b>	<b>6</b>
2.1 Policy . . . . .	8
2.2 Cumulative Return . . . . .	8
2.3 Value Function . . . . .	8
2.4 Bellman Equation . . . . .	9
2.5 Model-Free vs Model-Based Methods . . . . .	10
2.6 Q-Learning . . . . .	10
2.7 Policy Gradient . . . . .	11
2.8 Deep Deterministic Policy Gradient - DDPG . . . . .	12
2.9 Prioritized Experience Replay - PER . . . . .	12
<b>3 Neural Networks</b>	<b>14</b>
3.1 Building Blocks of ANNs . . . . .	14
3.1.1 Artificial Neuron . . . . .	14
3.1.2 Activation Functions . . . . .	16
3.1.3 Training the network . . . . .	17
<b>4 Prioritized Experience DDPG for Dynamic Systems with Continuous State Space</b>	<b>19</b>
4.1 The Contribution of This Work . . . . .	20
4.2 Proposed Method . . . . .	21
4.2.1 Prioritized Experience - Replay . . . . .	22
<b>5 Simulations and Results</b>	<b>25</b>
5.1 Inverted Pendulum Environment . . . . .	25

---

5.2	Learning Parameters . . . . .	26
5.2.1	Neural Network Parameters . . . . .	26
5.3	Results and Discussion . . . . .	27
5.4	Experiments on Algorithm Parameters . . . . .	29
<b>6</b>	<b>Conclusions and Future Work</b>	<b>32</b>
	<b>Bibliography</b>	<b>33</b>



# List of Figures

2.1	Machine Learning Approaches . . . . .	6
2.2	The interaction between agent and environment in MDP . . . . .	7
2.3	Model-free Algorithms . . . . .	10
3.1	Simple Feed Forward NN Example . . . . .	15
3.2	Simple CNN Neuron Example . . . . .	15
3.3	RNN Example . . . . .	16
3.4	Rectified Linear Unit . . . . .	17
3.5	Sigmoid Function . . . . .	17
3.6	Tanh Function. . . . .	17
5.1	Inverted Pendulum Simulation Snapshot . . . . .	26
5.2	Total Reward Per Episode of DDPG PE-DDPG . . . . .	28
5.3	Comparison of Sum of Absolute TD Error per Episode . . . . .	30
5.4	Comparison of Replay Buffer Size in PE-DDPG . . . . .	30
5.5	Comparison of Minibatch Size in PE-DDPG . . . . .	31

# List of Tables

5.1	State (Observation) space variables and ranges . . . . .	25
5.2	Action space variables and ranges . . . . .	26
5.3	Mean Cumulative Reward of Episode Intervals . . . . .	28
5.4	Standard Deviation of Cumulative Reward of Episode Intervals . . . . .	28
5.5	Total of Absolute TD-error of Episode Intervals . . . . .	29
5.6	Standard Deviation of TD-Error of Episode Intervals . . . . .	29

# Chapter 1

## Introduction

In this thesis, the sample efficient deep deterministic policy gradient method is proposed. This method aims to use deep learning methods to dynamic systems with continuous state space. It has two important properties: To decrease the learning time by making use of the experience better, and to solve the problem of correlated sequential experiences drawn with fast sampling from a dynamic system. In this method, rather than uniform sampling, prioritized experience sampling method is used. The method section describes the comparison between conventional DDPG [1] and our proposed method prioritized experience replay with rank based prioritization and proportional prioritization

### 1.1 Literature Review

One of the first successful learning methods is TD-Gammon [2], which was developed for the game backgammon and achieved super-human level of play by learning entirely with reinforcement learning and self-play. TD-Gammon algorithm was a model-free reinforcement learning and approximated the state value function  $V(s)$  rather than the action-value function  $Q(s,a)$ . After the implementation of the agent for backgammon, there were several attempts to solve at super-human levels in some games such as Go, checkers, chess. However, due to the randomness of dice rolls in backgammon, agents are able to explore the full state space in backgammon, unlike the other games. [3]

Although reinforcement learning agents achieved successful results in different domains like TD-gammon, these agents previously had some limitations specific to their domains, because their success were based on handcrafted features changing from domain to domain. In order to achieve success for different domains with one algorithm, model-free reinforcement algorithms have been proposed. One of the most important algorithms,

called Q-learning, is a model free algorithm and the agent is able to achieve success without a specific model designed for that domain with adjustable parameters [4].

Despite their success with arbitrary problem domains, when model-free reinforcement learning algorithms such as Q-learning are implemented using non-linear function approximators, the divergence problem might occur [5]. For that reason, researchers have focused on linear function approximations of model-free reinforcement learning algorithms and were limited to domains that have low dimensional input space.

The amount of research on the implementation of the RL algorithms using non-linear function approximators has recently been increasing. The neural network implementation of Q-learning has been investigated in prior works. Martin Riedmiller [6] proposed Neural Fitted Q-learning (NFQ) to optimize sequence of loss functions and update the parameters of Neural Network of Q-learning called Q-Network using Resilient Backpropagation (RPROP) algorithm. Although NFQ algorithm was one of the first attempts on non-linear approximation of Q-learning, the RPROP algorithm uses batch update whose computation cost increases when the amount of the data increase. Despite of computational cost, the NFQ algorithm was applied successfully to simple real-world control tasks using purely visual input, using autoencoders and applying RPROP algorithm update. Although, the NFQ algorithm can be considered as one of the earliest examples work of state-of-the-art deep reinforcement learning algorithms, previously, Q-learning has also been combined with simple neural network working on just low dimensional state rather than high dimensional input data [7]. Since the neural network in this algorithm [7] is a multi-layer perceptron, a weight change in certain part of states might cause a change of the values in other regions. This leads to very long time to learn or even to failure. Therefore, deep representation of reinforcement learning would need to be worked on higher dimensional states and learned in smaller time. Recent methods are able to solve high dimensional state and make an effort to decrease learning time.

Recent advancements on neural network architectures including convolutional neural networks [8], restricted Boltzmann machines [9] and recurrent neural networks [10],[11] allow deep reinforcement learning to successfully learn high dimensional states with end-to-end system. Mnih et.al proposed a deep learning model to achieve successfully learning control policies from high dimensional states. Basically, the model is an end-to-end architecture which composed of a convolutional neural network and a variant of Q-learning called Double Q-Network (DQN) [12]. DQN was applied to seven Atari 2600 games and achieved better than human results in three Atari games and outperforms all previous algorithms on six of Atari games.

Mnih et.al [13] proposes a more advanced version of DQN stated in [12], achieving a comparable level of a professional human game tester on 49 games. They claimed that

DQN is the bridge between high dimensional sensory input and actions. The learning agents are able to achieve successful learning in a diverse array of challenging tasks, with the same algorithm, network architecture and hyperparameters. The DQN algorithm also solves the instability problem or divergence problem by using experience replay buffer and updating the action-values toward target values only periodically. Those two main problems of non-linear approximation implementation of reinforcement learning by neural network occurred because of the correlation of experience data, however, two improvements in DQN stated in [13] reduce the correlations on experimental data and target action value and thus, better use of them.

Although DQN algorithm achieved performance comparable to the level of a professional human game tester, there were some games of Atari 2600 that the algorithm suffers from substantial overestimations of action values. In order to overcome overestimations, van Hasselt et. al, proposed a modification of Deep Q-Network called Double Deep Q-Network (DDQN) that uses the existing architecture and deep neural network of DQN algorithm without requiring additional networks or parameters [14]. Van Hasselt et.al, state that DQN causes overestimations because of using the same value both to select and evaluate an action by the max operator in Q-learning. However, this makes it more likely to select an overestimated action value. The proposed algorithm DDQN solve this problem by decoupling the selection from evaluation. According to the results of DDQN in Atari 2600 games, the algorithm does not reduce the observed estimations. It also leads to much better performance on several other games.

All the mentioned methods above except TD-Gammon were based on Q-learning which is a model-free algorithm and estimates discrete values of a value function  $Q$ , and tries to select a single deterministic action from a discrete set of actions by finding the maximum value (described later). However, there is another approach called “Policy Based Methods” in reinforcement learning, to maximize a cumulative future reward by learning a parametric map from state to action. With policy based approach, the learning problem becomes an optimization problem that tries to find best map from state to action: parameterized policy. Policy can work on continuous state and action spaces and can be stochastic, which is difficult to achieve for Q-learning that works on discrete action and state spaces. Q-learning applied to continuous action state spaces, it needs to discretize the continuous spaces. This would lead to the curse of dimensionality.

In policy based approach, there are several methods that have achieved successful results. Basically, the proposed successful algorithms can be gathered in three categories: Finite-Difference Methods, Monte-Carlo Policy Gradients and Actor-Critic Policy Gradients [15]. Finite-difference methods are among the oldest policy gradient algorithms that came into existence from the stochastic optimization community and are much easier to

understand compared to other approaches [16]. Spall et.al. [17] showed that this approach can be highly efficient in the simulation optimization of deterministic systems and Sehnke et.al, showed newer methods developed in this approach [18]. Since the method has originated from simulation and tries to optimize policy parameters with small variations, the algorithms are dependent on choosing initial value of policy parameters, and therefore, more likely to diverge when applied to real world dynamic problems. The Monte-Carlo Policy Gradient approach such as REINFORCE algorithm [19] estimates the policy gradient using the likelihood ratio. Since the estimation is based on likelihood ratio assuming the trajectory of policy generated from a system by roll-out, the estimates are unbiased compared to actor-critic policy gradients. Since the generation of policy parameter variations is no longer needed as opposed to finite-difference approach, possible problem of policy gradient parameters cannot endanger the policy estimation process. This approach is basically based on single roll-outs to produce unbiased estimate of policy gradient and also more applicable to real world robotics scenarios. Schaal et.al. states that the likelihood ratio is guaranteed to converge fastest among the other policy gradients for stochastic systems [20]. However, when used with deterministic policy and for continuous states and actions, the performance of the likelihood algorithm is not sufficient. Even the finite-difference approach which is the simplest approach among the policy gradient approaches is superior to Monte Carlo Policy Gradient approach. According to David Silver, Monte Carlo Policy Gradient approach has high variance and in order to decrease variance, critic function that estimates action value and evaluates policy, can be used [15]. This has recently lead researchers to actor-critic methods on policy gradients algorithms. Basically actor-critic algorithms are composed of two parts: critic and actor. The critic part is responsible for updating value function and the actor part is responsible for updating policy parameters in the direction suggested by critic. (Deeper detailed explanation of actor-critic methods can be found in later parts.) Actor-critic methods provide better converge properties than other policy gradient approaches due to the variance reduction by critic [21]. Also, learning is more stable than pure policy gradients such as Monte Carlo Policy Gradients and Finite-Difference Methods. For small state-spaces, tabular Temporal Difference algorithms to estimate Q-function such as SARSA [22], Q-learning can be used, however, for large state-spaces, neural network representations of both actor and critic part of the algorithms can be used as actor network and critic network separately. Good examples for neural network representation of actor-critic networks can be found in [23], [24].

Neural networks can be used to approximate the value function, the policy and the model as non-linear which is preferred in reinforcement learning. In [12], Q-learning, the value function, is approximated non-linearly by neural networks called Q-network. As proposed in [1] deep deterministic policy gradient, DDPG, the value function and

---

the policy function approximated by neural networks. However, direct implementation of neural network as a replacement of the functions of reinforcement learning components can be difficult and may even be futile. For example, if Q-learning function is replaced by only a Q-network, the implementation would be sufficient in theory, but, the algorithm provides a biased estimate causing algorithms to diverge. There are two main problems causing non-convergence. In training, a neural network expects to learn all different samples or inputs and assumes each input is different to each other, this means, it expects uncorrelated inputs. However, in reinforcement learning, [25] states that reinforcement learning agent experience are very correlated due to the underlying Markovian Decision Process. Therefore it is necessary to break correlations between states. This can be done by using a replay buffer which stores transitions or experiences off-line not online, and then neural network can use these experiences by sampling from the replay buffer. In DQN algorithm, using replay buffer solves the correlation problem. Another problem of neural network representation in reinforcement learning algorithms is that the target that the agent tries to reach is non-stationary. DQN algorithms solves this challenge by training the network with a target Q-network to give consistent targets during temporal difference backups [14]. Besides the successful non-linear approximation of policy gradient algorithm, David Silver's work gives good evidence that deterministic policy gradients can achieve better results compared to stochastic counterparts by several orders of magnitude in a bandit with 50 continuous action dimensions, and can solve a challenging reinforcement learning problem with 20 continuous action dimensions and 50 state dimensions [26].

## Chapter 2

# The Reinforcement Learning Problem

In this section, we start out with introducing the main concepts of reinforcement learning. We then go on to explain the base algorithm Deep Deterministic Policy Gradient and the Prioritized Experience Replay techniques are introduced.

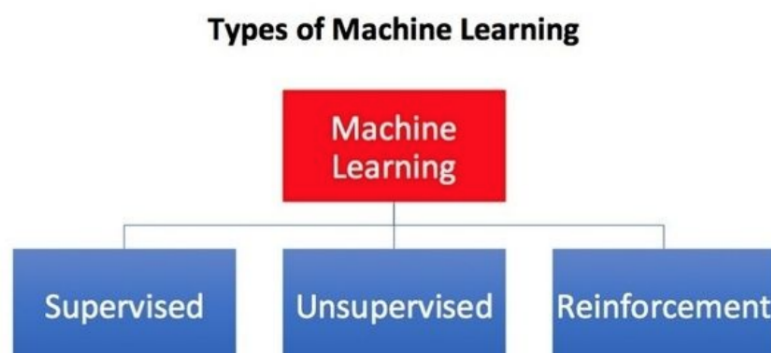


FIGURE 2.1: Machine Learning Approaches

Reinforcement Learning (RL) is a machine learning approach which differs from supervised learning in the feedback mechanism. Supervised learning is instructive in that, the agent is told how to achieve its goal, whereas in reinforcement learning the feedback is evaluative; only an instantaneous scalar signal that tells how well the agent is performing in its quest to achieve its goal is provided. In many machine learning scenarios, the evaluative feedback is more intuitive and accessible. Another difference is that RL agent finds optimal policy using trial and error, but supervised learning needs ground



truth value that the agent gets instructions. Therefore, for most of the control tasks, reinforcement learning gives outstanding results. Unlike these two learning approaches, in unsupervised learning which is another branch of machine learning, the agent only learns the structure of the data whereas RL can work with temporal state transitions.

As seen in Fig.2.2, the reinforcement learning problem consists of an agent and an environment modeled as a Markovian Decision Process (MDP). The agent acts on the environment and attains a scalar reward from environment that implies how well agent's action is. The main goal in reinforcement learning is to learn an optimal policy  $\pi$  which is a map from states to actions by maximizing the expected sum of rewards. RL algorithms assume the interaction between the agent and environment is sequential. Therefore, the reinforcement learning can be defined as sequential decision making. The sequential interaction between agent and environment is formulized as an MDP.

The other components of reinforcement learning are the policy that maps states to actions, the value function that gives the measurement of how well agent takes an action, and the Bellman Equation which allows recursive relationships in value function, will be explained in the following sections in this chapter.

An (MDP) is defined by a set of states,  $s \in S$ , where  $S$  denotes the state space, a set of actions,  $a \in A$ , where  $A$  denotes the action space, a scalar reward,  $r$ , discount factor,  $\gamma$ , and transition probability which is defined by

$$p(s'|s, a) = Pr(s_{t+1} = s' | s_t = s, a_t = a) \quad (2.1)$$

and the reward function,  $R : S \times S \times A \rightarrow \mathbb{R}$  which is defined by

$$R(s, a, s') = \mathbb{E}(r_t | s_t = s, a_t = a, s_{t+1} = s') \quad (2.2)$$

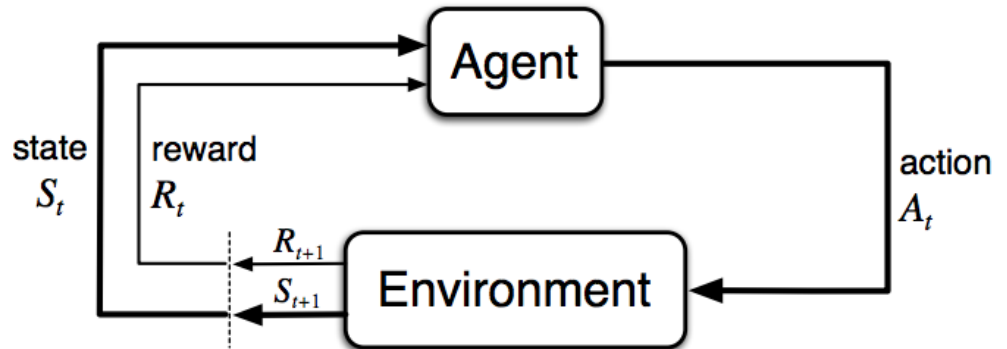


FIGURE 2.2: The interaction between agent and environment in MDP

## 2.1 Policy

The policy is defined as a mapping from states to actions. The policy could be deterministic, and depend only on the state,  $\pi(s)$ , or stochastic,  $\pi(a|s)$ , such that it defines a probability distribution over the actions, given a state.

## 2.2 Cumulative Return

The objective of reinforcement learning is to learn policy  $\pi : S \rightarrow A$  that maximizes the sum of the rewards. The return of a state can be measured as the weighted sum of the future rewards. The return of a state can be defined as:

$$R_t = r(s_t) + r(s_{t+1}) + r(s_{t+2}) + \dots r(s_{T-1}) \quad (2.3)$$

where,  $T$  denotes the terminal state that is the end of an episode,  $t$  is the time index.

For non-terminating tasks, the discounted return is defined, which is given by

$$R_t = r(s_t) + \gamma r(s_{t+1}) + \gamma^2 r(s_{t+2}) + \dots = \sum_{i=t}^T \gamma^{i-t} r(s_i) \quad (2.4)$$

where  $\gamma \in [0, 1)$  is called the discount factor.

## 2.3 Value Function

Value function measures how well an agent's action is. Value function is defined as the expected sum of rewards following some policy  $\pi$  from a particular state  $s$ . The value function,  $V_\pi(s)$  for policy  $\pi$  is given by

$$V^\pi(s) = \mathbb{E}_\pi(R_t | s_t = s) = \mathbb{E}_\pi\left(\sum_{i=t}^T \gamma^{i-t} r(s_i) | s_t = s\right) \quad (2.5)$$

Similarly, an *action-value function*, also known as the *Q-function*, can be defined as the expected sum of rewards while taking action  $a$  in state  $s$  and, thereafter, following policy  $\pi$ . The action-value function describes the expected return after taking an action  $a_t$  in state  $s_t$ , with the policy  $\pi$ :

$$Q^\pi(s_t, a_t) = \mathbb{E}_{r_{i \geq t}, s_{i > t} \sim E, a_{i > t} \sim \pi} [R_t, | s_t, a_t] \quad (2.6)$$

$$Q^\pi(s, a) = \mathbb{E}_\pi(R_t | s_t = s, a_t = a) = \mathbb{E}_\pi\left(\sum_{k=0}^{\infty} \gamma^k r_{t+k} | s_t = s, a_t = a\right) \quad (2.7)$$

## 2.4 Bellman Equation

The Bellman equations allow us to solve MDP problems in reinforcement learning setup by formulating the problem of maximizing the expected sum of rewards in terms of recursive relationship of a value function  $V(s)$ . A policy  $\pi$  is considered better than another policy  $\pi'$  if the expected return of that policy is greater than  $\pi'$  for all  $s \in S$ , which implies,  $V^\pi(s) \geq V^{\pi'}(s)$  for all  $s \in S$ . Thus, the optimal value function,  $V^*(s)$  can be defined as,

$$V^*(s) = \max_{\pi} V_{\pi}(s), \forall s \in S. \quad (2.8)$$

Similarly, the optimal *action – value function*,  $Q^*(s, a)$  can be defined as,

$$Q^*(s) = \max_{\pi} Q_{\pi}(s, a), \forall s \in S, a \in A. \quad (2.9)$$

Also, for an optimal policy, the following equation can be written,

$$V^*(s) = \max_{a \in A(s)} Q_{\pi^*}(s, a) \quad (2.10)$$

Expanding (2.10) with (2.7),

$$\begin{aligned} V^*(s) &= \max_a \mathbb{E}_{\pi^*}(R_t | s_t = s, a_t = a) \\ &= \max_a \mathbb{E}_{\pi^*}\left(\sum_{i=t}^T \gamma^{i-t} r(s_i) | s_t = s\right) \\ &= \max_a \sum_{s'} p(s' | s, a) [R(s, a, s') + \gamma V^*(s')] \end{aligned} \quad (2.11)$$

where  $s'$  and  $a'$  are possible next value of current state  $s$  and action  $a$  respectively.

Equation (2.11) is known as the Bellman optimality equation for  $V^*(s)$ . The Bellman optimality equation for  $Q$  can be recursively written as

$$\begin{aligned} Q^*(s, a) &= \mathbb{E}(r_t + \gamma \max_{a'} Q^*(s', a') | s_t = s, a_t = a) \\ &= \sum_{s'} p(s' | s, a) [R(s, a, s') + \gamma \max_{a'} Q^*(s', a')] \end{aligned} \quad (2.12)$$

After taking an action, the agent observes its outcome and updates the  $Q$  values, using (2.12) in order to approach the optimal policy  $\pi^*$ .

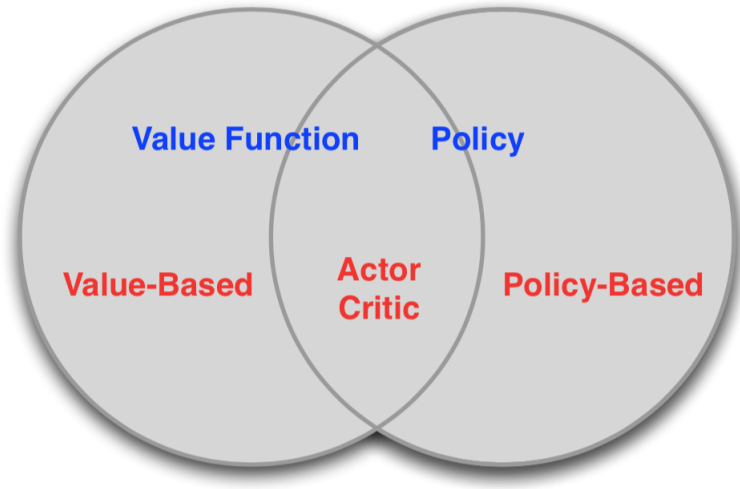


FIGURE 2.3: Model-free Algorithms

## 2.5 Model-Free vs Model-Based Methods

If all the components of MDP including (2.1) and (2.2) are known by the agent, there is no need to observe agent's action and evaluate action using state-action value function  $Q(s, a)$ . Then, the problem turns into a planning problem and can be solved with iterative methods.

However, reinforcement learning problem is different than planning that agent does not know all the components of the MDPs. Model in reinforcement learning can be thought of agent's own representation of environment which the agent interacts. Model free algorithms relies only on experiences of agent, and the agent updates its policy after each experience. The agent updates its policy after getting reward as shown in Figure 2.2. However, model based algorithms constructs its model based on experiences, and after constructing a model, the agent simulate further episodes. Thus, learning in model based can be shorter than model free algorithms. However, if model based algorithms learn the model incorrect, the whole learning process might be different than reality.

## 2.6 Q-Learning

Q-learning is a value based model-free reinforcement learning algorithm where main goal is to find optimal action selection policy using a Q value function, as stated in 2.7. According to [25], Q-learning makes the agent learn acting optimally in finite MDPs without any domain knowledge. Watkins and Dayan [4] proved that Q Learning

converges if all actions are experienced by agent in all states by assuming the action values are represented discrete. Q learning can be considered as an off-policy algorithm, because, the actions for Q function are taken disregarding the current policy such as taking actions randomly. The agent in Q-learning seeks to learn a policy that maximizes the cumulative return.

In Q learning, the value of each experience is stored the table that is a matrix with the dimensions of the number of states by the number of actions with the initial value of zero. Q value of each state action pair is stored in Q-table after the agent finishes an episode. The agent chooses its actions according to Q values in Q-table. The agent needs to explore the environment before it exploits from it's experiences. In the exploration step, the agent chooses actions randomly, however, in exploitation, the actions are chosen by maximizing the Q function. After an observation, the Q value is updated by:

$$Q^*(s_t, a_t) = Q(s_t, a_t) + \alpha(r(s_t, a_t) + \gamma \max_{a_{t+1}} Q(s_{t+1}, a_{t+1})) \quad (2.13)$$

where  $\alpha$  is learning rate which decreases slowly.

## 2.7 Policy Gradient

Policy gradient algorithms are widely used in reinforcement learning for continuous state and action spaces. Policy gradient turns the reinforcement learning problem into a search for the optimal parameters  $\theta$  of the policy ( $\pi_\theta$ ) that maximize the objective function  $J(\theta)$ . As stated in Section 2.1, policy  $\pi_\theta(s)$  can be deterministic which outputs exact action to be taken or stochastic which outputs the probability of each actions that can be taken. As stated in [26], in policy gradients, the goal of an agent is to find policy which maximizes the cumulative discounted reward from the start state. The objective function can be given by:

$$J(\pi) = \mathbb{E}[R_t | \pi] \quad (2.14)$$

According to [26], equation 2.14 can be rewritten with respect to the optimization parameter  $\theta$  as:

$$\begin{aligned} J(\pi_\theta) &= \int_S \rho^\pi(s) \int_A \pi_\theta(s, a) r(s, a) da ds \\ &= \mathbb{E}_{s \sim \rho^\pi, a \sim \pi_\theta} [r(s, a)] \end{aligned} \quad (2.15)$$

where  $\rho^\pi(s)$  is a discounted state distribution under the policy  $\pi$ .

In order to find optimal policy, parameter  $\theta$  is supposed to be adjusted in the direction of gradient of the objective function  $J(\pi_\theta)$ . The gradient of the objective function can be given by:

$$\begin{aligned}\nabla_\theta J(\pi_\theta) &= \int_S \rho^\pi(s) \int_A \nabla_\theta \pi_\theta(a|s) Q^\pi(s, a) da ds \\ &= \mathbb{E}_{s \sim \rho^\pi, a \sim \pi_\theta} [\nabla_\theta \log \pi_\theta(a|s) Q^\pi(s, a)]\end{aligned}\tag{2.16}$$

Equation (2.16) is the main algorithm for the policy gradient theorem in reinforcement learning.

## 2.8 Deep Deterministic Policy Gradient - DDPG

The Bellman Equation (Sec 2.4), becomes hard to calculate when action and state spaces are continuous. One solution could be discretizing the continuous spaces, however, this method causes the curse of dimensionality. To overcome this problem, there is a need to calculate the state-action function continuously rather than discretizing. Neural Network approach can calculate the Bellman Equation by using two networks, i.e one for calculating action values and one for calculating Q values. This model is based on actor-critic approach by applying non-linear approximation of Bellman Equation. As stated in [1], the model free actor-critic method called DDPG is based on the deterministic policy gradient algorithms that proposed by Silver et. al [26].

In [1], the authors claimed that DDPG method is a combination of actor-critic approach and one of the important milestone algorithm called Deep Q Network which is discrete neural network representation of Bellman function proposed in [4]. Lillicrap et al. states that model-free approach called Deep Deterministic Policy Gradient can learn competitive policies for all of tasks using low-dimensional observations using the same hyper-parameters and network structure.

## 2.9 Prioritized Experience Replay - PER

As mentioned earlier, reinforcement learning algorithms use uniform sampling from experience replay buffer in order to prevent itself from correlated samples. However, prioritized experience replay technique proposes a method that sampling an experience in a smarter way. As [27] states that each experience could not contribute equally, therefore, this method aims to sample experiences that reinforcement policy is not good at, more than sampling experiences that the policy is good at. Specifically, Schaul et.al [27],

proposes to more frequently replay transitions with high expected learning progress, as measured by the magnitude of their temporal difference (TD) error. The transitions are described as the atomic unit of interaction in Reinforcement Learning, in our case as tuple of (state  $s_{t-1}$ , action  $a_{t-1}$ , reward  $r_t$ , discount factor  $\gamma_t$ , next state  $s_t$ ). However, sampling transitions with high Temporal Difference (TD) Errors which is the difference between target Q value and recent Q value, more frequently causes the loss of diversity and introducing diversity. Schaul et al solve this problem using stochastic prioritization and bias can be corrected by importance sampling.

When the agent starts learning, there would be a lot of transitions with high TD error, and it means that initially high error transitions get replayed frequently. This problem is the result of the lack of diversity as mentioned above. In order to overcome this issue, Schaul et al introduces a stochastic prioritization technique that interpolates between pure greedy prioritization and uniform sampling. The stochastic prioritization will be discussed in Chapter 4.2.1. The prioritization technique is not always sampling with TD Error. [27] states that there is a balancing mechanism between pure greedy prioritization and uniform sampling.

Also there are two variants of Prioritized Experience replay: the proportional prioritization and rank based prioritization. This research used proportional prioritization that will be discussed later chapter.

## Chapter 3

# Neural Networks

Artificial Neural Networks (ANN) are computing systems that are inspired by the biologic neural networks and loosely mimic the way of processing information in biological neural networks in the human brain. Neural Networks are function approximators whose tasks are to recognize patterns. The patterns are numeric values such as images, sound, text or time series. Thanks to many breakthrough results, ANNs are at the most important components of some systems in self-driving cars [28], image recognition systems [29], speech recognition systems [30], recommender systems [31] and autonomous robots [32]. ANN works as non-linear function approximators to map input to output.

### 3.1 Building Blocks of ANNs

ANNs consist of several building blocks. Their functions are described in next sections.

#### 3.1.1 Artificial Neuron

Biological brains have many connections and components to transfer processed information from one unit to another. Likewise, ANNs also have connections between ANN units, and some functions in units in order to process information. Basic unit in ANN is a neuron. The major components of neurons are weights, bias and activation functions. Processing an information is generally done according to the function:

$$y = f\left(\sum_i x_{ij}w_{ij} + b_j\right) \quad (3.1)$$

where  $x_{ij}$  is the input of each neuron from the previous layer,  $w_{ij}$  is the weight between neurons,  $b_j$  is the bias value in neuron,  $f$  is the activation function of a neuron and,  $y$  is



the output. A neural network made up several hidden neurons and one output neuron can be seen in Figure 3.1.

Artificial neural networks architectures can be classified with respect to the connections between neurons such as Feed Forward Neural Networks, Convolutional Neural Networks (CNN) and, Recurrent Neural Networks (RNN). The neurons of Feed Forward Neural Networks are directly connected to each other such that the outputs of neurons in one layer are connected to the inputs of neurons in next layer. Each neuron receives input from every neuron of the previous layer.

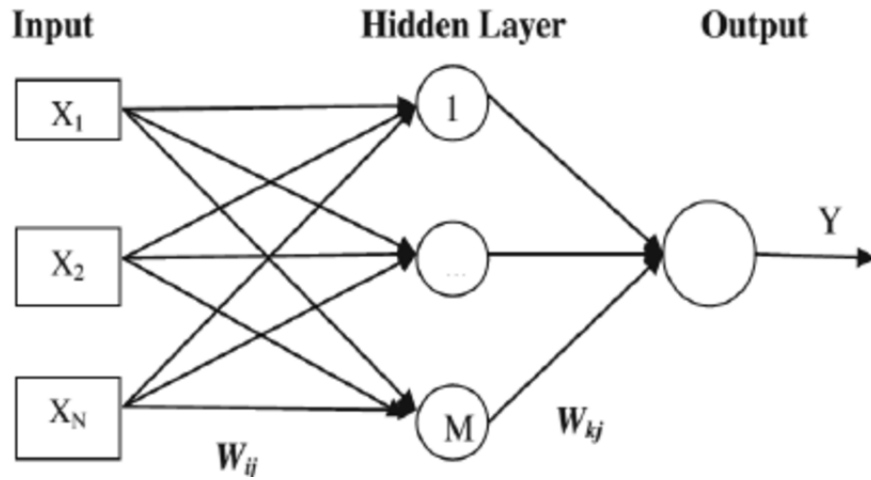


FIGURE 3.1: Simple Feed Forward NN Example

Neurons in Convolutional Neural Networks receive input from only a restricted subarea of the previous layer. The CNN is mostly applied to image recognition tasks. Inputs are 2 or 3 dimensional matrices of numbers.

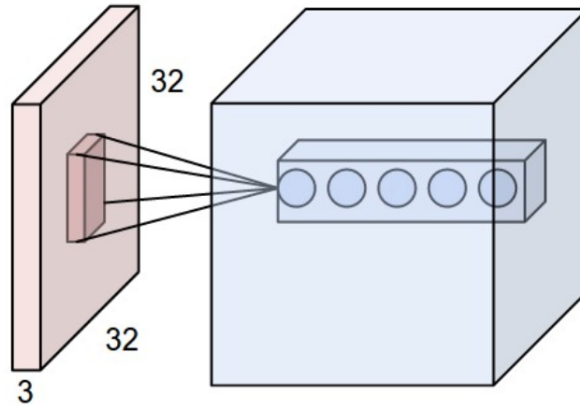


FIGURE 3.2: Simple CNN Neuron Example

The Figure 3.2 shows the connection between input layer (on the left) and the first layer of CNN (on the right).

The neurons in RNN on the other hand, receive input from neurons of previous layer and from its own output itself, as shown in Fig.3.3. RNN has a feedback loop from the output of the recurrent neurons to their inputs. RNN displays outstanding performance in natural language processing. Training of RNN is more difficult since it must learn a state transition function, on top of the previous input output relationship. The state transition function is generally not supervised.

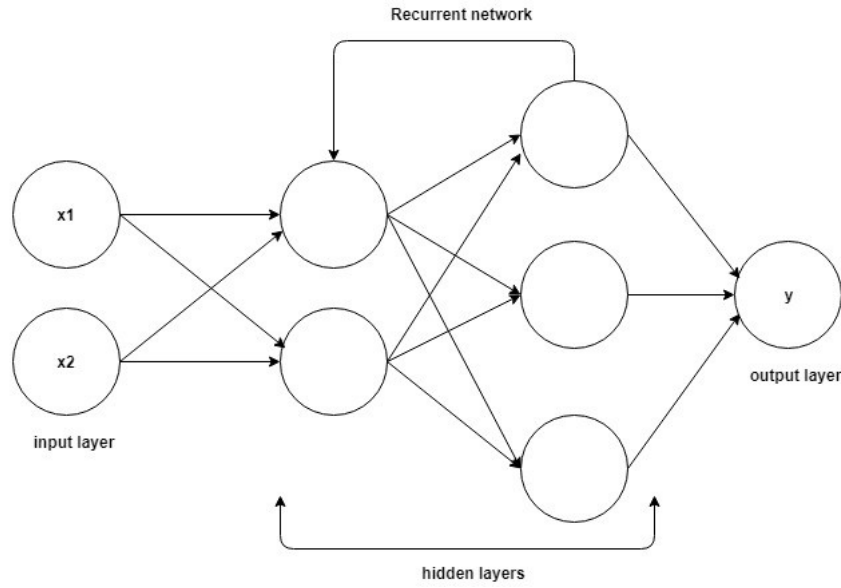


FIGURE 3.3: RNN Example

### 3.1.2 Activation Functions

As mentioned before, neural networks are used in approximating non-linear functions. Activation functions introduce non-linearity to neurons. Most commonly used activation functions are rectified linear unit (ReLU), sigmoid function and hyperbolic tangent (tanh).

ReLU function shown in Fig.3.4 is  $y = f(x) = \max(0, x)$ , mostly used in CNN architectures. The range of ReLU output is  $[0, \infty]$ .

The sigmoid function shown in Fig.3.5 is  $y = f(x) = \frac{1}{1+e^{-x}}$ . It is commonly used in classification, making clear distinctions on prediction.

The hyperbolic tangent function shown in Fig.3.6 is  $y = f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$ . Tanh is also used in classification commonly.

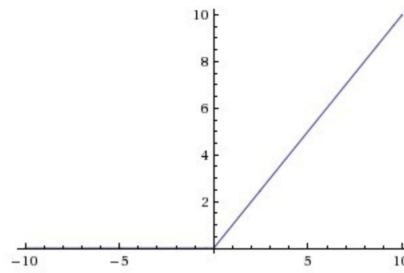


FIGURE 3.4: Rectified Linear Unit

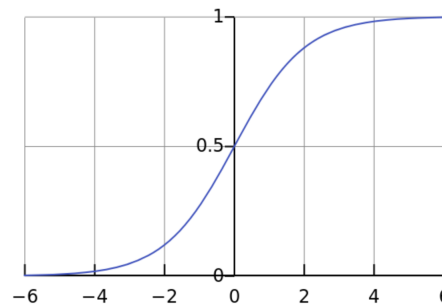


FIGURE 3.5: Sigmoid Function

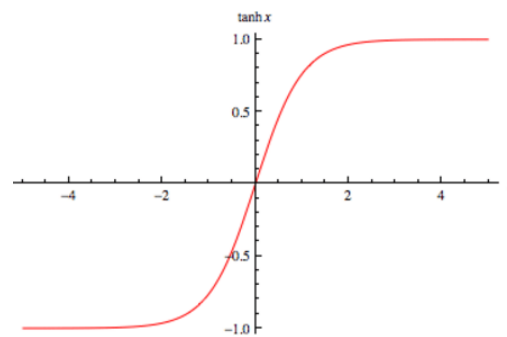


FIGURE 3.6: Tanh Function.

### 3.1.3 Training the network

Ultimate goal of training the neural network is to update the network parameters weights and biases in order to minimize the error between generated output of network and the desired output. Neural networks need a cost function  $J(\theta)$  where  $\theta$  represents all of the network parameters, in order to measure the error between generated output  $o$  and the desired output  $y$ . All weights are then updated using an algorithm that propagates the error into the weights of each layer, such as backpropagation. This is commonly called the training algorithm.

Since deep learning methods need large amounts of data, the stochastic gradient descent algorithm is commonly used in backpropagation updates of weights.

$$w_{ij}(t+1) = w_{ij}(t) + \eta \frac{\partial J(\theta)}{\partial w_{ij}} \quad (3.2)$$

where  $w_{ij}(t+1)$  is the updated value of the weight, and  $w_{ij}(t)$  is the value of the same weight before update,  $\eta$  is the learning rate and the last term is the partial derivative of cost function with respect to weight  $w_{ij}$  that gives the rate of cost function change with weight. In stochastic gradient descent, weights of networks are updated just after calculation of cost value from one input. Therefore, stochastic gradient descent is very popular in deep learning methods. Some improvements have been proposed e.g., Adam [33], and AdaGrad [34] where the learning rate  $\eta$  changes based on the distribution of the input data. This improvement makes the neural network converge faster.

To sum up, various steps in in each iteration of learning are:

1. Select a network architecture by determining the number of hidden layers, the number of neurons in each layer, and the activation functions in each neuron,
2. Assign random initial values to network parameters
3. Propagate the network forward to get output in iteration
4. Calculate the error in determined cost function,
5. Backpropagate the network finding the error for each neuron given actual output and desired data,
6. Update the weights with errors to minimize cost function, and repeat from Step 3.

## Chapter 4

# Prioritized Experience DDPG for Dynamic Systems with Continuous State Space

Deterministic policy gradients achieve more successful results than stochastic counterparts in policy gradient algorithms. In this research, we propose an algorithm for deep deterministic policy gradients with prioritized experience replay in order to improve results compared to stochastic gradients and we planned to decrease the number of required samples for completely learning the system by using the technique of prioritization.

The earliest works showed that the use of large, non-linear function approximators for action-value functions causes instability and can not give theoretical performance guarantee. With the major changes introduced by DQN algorithm, the use of large neural networks as effective function approximators becomes possible: the use of a replay buffer that makes the learning algorithms work as off-policy where agent uses samples from past experience rather than current experience and a separate target network for calculating the target value  $y_t$ . Lillicrap et.al [1] employed these into Deep Deterministic Policy Gradient(DDPG) algorithm. We propose Prioritized Experience DDPG (PE-DDPG) method in this thesis, which integrates DDPG with smarter experience replay.

The DPG algorithm proposed by Silver et.al [26] , introduces a parameterized actor function  $\mu(s|\theta^\mu)$  which specifies the current policy rather than using greedy policy  $\mu(s) = \operatorname{argmax}_a Q(s, a)$ . The policy (action function)  $\mu(s|\theta^\mu)$  deterministically maps a state to a specific action.

The proposed PE-DDPG algorithm has two networks; actor and critic. Actor collects information related to greedy policy and critic updates policy with the information

received from actor network. There are two learning phases in the algorithm for each. Learning of the critic  $Q(s, a)$  can be done by using the Bellman equation as in Q-learning. The actor network is updated using the chain rule of the expected return from the start distribution  $J$  with respect to the actor parameters:

$$\begin{aligned}\nabla_{\theta^\mu} J &\approx \mathbb{E}_{s_t \sim \rho^\beta} [\nabla_{\theta^\mu} Q(s, a | \theta^Q) |_{s=s_t, a=\mu(s_t | \theta^\mu)}] \\ &= \mathbb{E}_{s_t \sim \rho^\beta} [\nabla_a Q(s, a | \theta^Q) |_{s=s_t, a=\mu(s_t)} \nabla_{\theta^\mu} \mu(s | \theta^\mu)_{s=s_t}]\end{aligned}\tag{4.1}$$

When neural networks are used for reinforcement learning, one of the most important challenging problems is correlation between experiences. Since most optimization algorithms in neural networks assume that the samples are independently and ideally distributed, it is impossible to apply neural network to reinforcement learning algorithm where the samples are generated sequentially from the environment. According to [27], minibatch updates can be used to make efficient use of hardware optimizations rather than updating online.

The PE-DDPG method proposed in this thesis, uses replay buffer to break correlations between experiences as in original DDPG algorithm. The replay buffer is a finite sized data collection  $H$ . In the replay buffer, transition tuples  $s, a, r, s'$  are stored. At each time step, as it can be seen on Algorithm 1, the agent with the algorithm gets mini-batch of samples with the size  $N$ .

## 4.1 The Contribution of This Work

We proposed to change the way of sampling transitions from the replay buffer compared to the original DDPG algorithm. In original DDPG, uniform sampling is used and this makes all transitions for the agent equally important. However, this is not true in reality, because, the transitions that have high TD-error should have more effect on learning. It means the agent is not good at those transitions. Also, for the transitions with low TD-error, gradient would not be high and the amount of update is not sufficient. Therefore, resampling these experiences would not contribute learning, but sampling those transitions with high TD-error would contribute to learning more. If agent sees these transitions more often compared to transitions with low error, the algorithm would learn more efficiently. The TD-error can be thought of as an indicator of how surprising or unexpected the transition is.

As stated in prior chapters, the DQN algorithm showed that non-linear approximation of reinforcement learning algorithms would give good result by avoiding the divergence problem. However, DQN algorithm works on discrete state space environments and

the only way of using DQN for continuous space is to discretize the continuous space. However, there are some limitations to the discretization of continuous domains on the algorithm performance. The most notable limitation is the curse of dimensionality where the number of actions increase exponentially with the number of degrees of freedom. The human arm can be a good example for this problem. The action space in the human arm as a 7 degree of freedom system, can be discretized in the coarsest way with  $a_i \in \{-k, 0, k\}$ . Since this action discretization would be applied to each joint, this leads to an action space with dimensionality:  $3^7 = 2187$  [1]. The tasks that need finer control of actions as they require a finer grained discretization makes this situation even worse, because, it leads to an explosion of the number of discrete actions. Therefore, successfully training of DQN-like networks in this context is likely intractable. In addition to the curse of dimensionality problem, naive discretization of action spaces may discard information about the structure of the action domain, which may be essential for solving many problems.

This work presents a model-free, where agent can learn different environments without any information about environment, off-policy actor-critic algorithm stated in [1] with sample efficiency with the help of the prioritization technique stated in [27].

## 4.2 Proposed Method

The complete PE-DDPG method can be seen in Algorithm 1. The main idea behind the approach is to achieve a faster learning algorithm working on dynamic environments with continuous state space such as inverted pendulum, by sampling the experiences that the agent is not good at, more. By sampling the experiences with higher error more frequently, the amount of experience or data need for reinforcement learning agent would decrease.

The policy  $\pi$  defines an agent's behaviour, and maps the states to a probability distribution over the actions, i.e.,  $\pi : S \rightarrow P(A)$ . The environment,  $E$  is a Markov Decision Process (MDP) model given by a tuple  $(S, A, r, p)$  where  $S$  is continuous state space  $S = \mathbb{R}^N$ ,  $A$  is continuous action space  $A = \mathbb{R}^N$ , initial state distribution is  $p(s_1)$ , transition dynamics are described by  $p(s_{t+1}|s_t, a_t)$ , and  $r$  is the reward function  $r(s_t, a_t)$ .

The action-value function describes the expected return after taking an action  $a_t$  in state  $s_t$ , with the policy  $\pi$ :

$$Q^\pi(s_t, a_t) = \mathbb{E}_{r_{i \geq t}, s_{i > t} \sim E, a_{i > t} \sim \pi} [R_t, |s_t, a_t] \quad (4.2)$$

Bellman equation allows to modify equation (4.2) into a recursive relationship as follows:

$$Q^\pi(s_t, a_t) = \mathbb{E}_{r_t, s_{t+1} \sim E}[r(s_t, a_t) + \gamma \mathbb{E}_{a_{t+1} \sim \pi}[Q^\pi(s_{t+1}, a_{t+1})]] \quad (4.3)$$

Q-learning posed as an off-policy learning algorithm in the proposed method, uses greedy policy  $\mu(s) = \operatorname{argmax}_a Q(s, a)$ . This deterministic policy can be explicitly written as a function and the action term in above equation can be rewritten as a function of the greedy policy  $\mu : S \rightarrow A$  :

$$Q^\pi(s_t, a_t) = \mathbb{E}_{r_t, s_{t+1} \sim E}[r(s_t, a_t) + \gamma Q^\mu(s_{t+1}, \mu(s_{t+1}))] \quad (4.4)$$

According to the equation above, the agent can learn off-policy because experience replay makes it possible to sample agent's past experience.

Also, the proposed algorithm uses function approximators parameterized by  $\theta^Q$  and  $\theta^\mu$ , the agent can learn by optimizing the following equation:

$$L(\theta^Q) = \mathbb{E}_{s_t \sim \rho^\beta, a_t \sim \beta, r_t \sim E}[(Q(s_t, a_t | \theta^Q) - y_t)] \quad (4.5)$$

where  $y_t = r(s_t, a_t) + \gamma Q(s_{t+1}, \mu(s_{t+1}) | \theta^Q)$

#### 4.2.1 Prioritized Experience - Replay

The proposed algorithm uses the stochastic sampling method that interpolates between pure greedy prioritization and uniform random sampling. The probability of sampling transition  $i$  is given by:

$$P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha} \quad (4.6)$$

where  $p_i > 0$  is the priority measure of transition  $i$ . The constant parameter  $\alpha$  controls how much prioritization is used and makes the prioritization interpolated between pure greedy and uniform sampling where  $\alpha = 0$ .

The algorithm uses proportional prioritization stated in [27] where  $p_i = |\delta_i| + \epsilon$ . Adding a small constant,  $\epsilon$ , prevents the transitions that have TD-error zero from not being revisited.

This interpolation also contributes to the reduction of resampling frequency of high error initial transitions, because error shrinks slowly when using function approximation, causing the lack of diversity making system to prone to over-fitting.



---

**Algorithm 1:** Prioritized Experience DDPG

---

**Input:** minibatch  $N$ , step-size  $\eta$ , replay period  $K$ , and replay buffer size  $Z$ , exponents  $\alpha$  and  $\beta$

Randomly initialize critic network  $Q(s, a|\theta^Q)$  with weight  $\theta^Q$ ,

Randomly initialize actor network  $\mu(s|\theta^\mu)$  with weight  $\theta^\mu$ ,

Initialize target network  $Q'$  and  $\mu'$  with weight  $\theta^{Q'} \leftarrow \theta^Q$ ,  $\theta^{\mu'} \leftarrow \theta^\mu$ ,

Initialize replay buffer  $H = \emptyset$ ,  $p_1 = 1$

**for**  $episode = 1, M$  **do**

    Initialize a random process  $N$  for action exploration

    Receive initial observation state  $s_1$

**for**  $t=1, T$  **do**

        Select action  $a_t = \mu(s_t|\theta^\mu) + N_t$  according to the current policy and exploration noise

        Execute action  $a_t$  and observe reward  $r_t$  and observe new state  $s_{t+1}$

        Store transition  $(s_t, a_t, r_t, s_{t+1})$  in  $H$

**if**  $t \parallel_k \equiv 0$  **then**

**for**  $j=1, N$  **do**

                Sample transition  $j \sim P(j) = p_j^\alpha / \sum_i p_i^\alpha$

                Compute importance-sampling weight  $w_j = (N * P(j))^{-\beta} / \max_i w_i$

                Compute TD-error  $\delta_j = r_j + \gamma Q'(s_{j+1}, \mu'(s_{j+1}|\theta^{\mu'})|\theta^{Q'}) - Q(s_j, a_j|\theta^Q)$

                Update transition priority  $p_j \leftarrow |\delta_j|$

**end**

        Update critic by minimizing the loss:  $L = \frac{1}{N} \sum_i w_i \delta_i^2$

        Update the actor policy using the sampled policy gradient:

$$\nabla_{\theta^\mu} J \approx \frac{1}{N} \sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s_i}$$

        Update target networks:

$$\theta^{Q'} \leftarrow \tau \theta^Q + (1 - \tau) \theta^{Q'}$$

$$\theta^{\mu'} \leftarrow \tau \theta^\mu + (1 - \tau) \theta^{\mu'}$$

**end**

**end**

**end**

---

Prioritized replay introduces a bias, because it changes the distribution of stochastic updates. Therefore, prioritization changes the solution that the estimate of the algorithm will converge to. This bias can be corrected using importance sampling weights as following:

$$w_j = \left( \frac{1}{N} \frac{1}{P(i)} \right)^\beta \quad (4.7)$$

where  $\beta$  is in  $[0, 1]$ . These weights are used in DDPG algorithm update as  $w_i \delta_i$  instead of  $\delta_i$ .

As stated in 1, the critic network is updated by minimizing the loss function. The weights of critic network are updated with the differentiation of the loss function with regard to its weights  $\theta^Q$ . Suppose one experience called  $c$  is chosen to update the critic network. The differentiation of loss function for weight  $\theta_1^Q$  can be computed as follows:

$$\begin{aligned}
\nabla_{\theta_1^Q} L(\theta_1^Q) &= \frac{1}{2} \nabla_{\theta_1^Q} (r_j + \gamma Q'(s_{t+1}, a_{t+1} | \theta_1^{Q'}) - Q(s_t, a_t | \theta_1^Q))^2 \\
&= (r_j + \gamma Q'(s_{t+1}, a_{t+1} | \theta_1^{Q'}) - Q(s_t, a_t | \theta_1^Q)) (\theta_1^Q) \nabla_{\theta_1^Q} Q(s, a | \theta_1^Q) \\
&= \delta_c \nabla_{\theta_1^Q} Q(s, a | \theta_1^Q)
\end{aligned} \tag{4.8}$$

Since PE-DDPG method replays experiments with high magnitudes of TD-Error, TD-Error  $\delta$  in the equation 4.8 has a large magnitude which leads critic network to be destabilized. Thus, importance sampling weights are applied to loss function in order to remove the effect of large updates on the network.

The proposed algorithm in this thesis normalizes the weights by  $1/\max_i w_i$ , in order to scale the update downwards as stated in [27].

Based on the prioritized experience replay equation, the PE-DDPG method; DDPG with prioritized experience replay, is proposed.

## Chapter 5

# Simulations and Results

The proposed Prioritized Experience DDPG (PE-DDPG) algorithm is tested on OpenAI's GYM environment. Gym is a toolkit for developing and comparing reinforcement learning algorithms. The toolkit has no assumptions about the structure of reinforcement learning agents.

In order to test PE-DDPG, the continuous space environments such as inverted pendulum of Gym toolkit are used.

### 5.1 Inverted Pendulum Environment

Inverted pendulum task is a classic control task in reinforcement learning. The aim is to keep a frictionless pendulum standing up, meaning that keep pendulum with zero angle. The reason why inverted pendulum is chosen is because the environment has continuous action and state spaces. The state and action ranges of the inverted pendulum plant are shown in Table 5.1

TABLE 5.1: State (Observation) space variables and ranges

Observation	Min	Max
$\cos(\theta)$	-1.0	1.0
$\sin(\theta)$	-1.0	1.0
$\dot{\theta}$	-8.0	8.0

All the variables can attain any real numbers between minimum and maximum values due to the continuous state space property of inverted pendulum simulation.

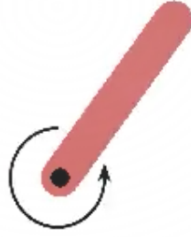


FIGURE 5.1: Inverted Pendulum Simulation Snapshot

TABLE 5.2: Action space variables and ranges

Action	Min	Max
Joint effort	-2.0	2.0

The reward function of inverted pendulum is:

$$r = -(\theta^2 + 0.1 * \dot{\theta}^2 + 0.001 * a^2)$$

In inverted pendulum environment of Gym toolkit,  $\theta$  is normalized to  $-\pi$  to  $\pi$ . According to environment variables value range stated above, the lowest value of reward is  $-16.2$  and highest value is 0. This means that the goal of the agent is to remain at vertical, with the least rotational velocity and the least effort. Since there is no termination point for the environment, each episode in the algorithm is bounded by the given maximum number of time steps.

## 5.2 Learning Parameters

### 5.2.1 Neural Network Parameters

We used momentum based Adam optimizer [33] for optimizing the loss in neural networks with a learning rate of  $10^{-4}$  and  $10^{-3}$  for the actor and critic networks respectively as

stated in [1]. To prevent large value of weights in critic network  $L_2$  weight decay [35] of  $10^{-2}$  is applied. A discount factor of  $\gamma = 0.99$  are used for Q function. For the soft updates  $\tau = 0.0001$  is used. In order to limit the action value that comes from actor network, the  $\tanh$  function is used at its output layer. Since the proposed method uses low dimensional input coming from toolkit environment, we use low-dimensional networks version as stated in [1] that have 2 hidden layers with 400 and 300 units for actor and critic respectively ( $\equiv 130000$  parameters). Actions are included Q-network at the 2nd hidden layer rather than the first hidden layer as direct input to network.

In order to sample from the replay buffer with respect to the priority of each experiment, sum tree data structure where leaf nodes have the value of each experiment's priority is used. In the sum tree of our sampling algorithm, top node has the value of the total of the priorities of each experiment  $p_{total}$  and internal nodes have intermediate sum. As stated in [27], sum tree data structure allows us to update and sample experiences in  $\mathcal{O}(\log N)$  operations. Experiences with top  $N$  priority values are stored in the replay buffer and those experiences are sampled by dividing equally the range  $[0, p_{total}]$  into  $k$  ranges where  $k$  is the minibatch size. Then uniform sampling is applied to each range in order to sample one experience from each range.

### 5.3 Results and Discussion

We tested PE-DDPG on OpenAI Gym's inverted pendulum environment which has continuous action and state spaces. All the experiments were done with 500 episodes where each episode has 200 iterations. The experiment compares two algorithms; DDPG [1] and PE-DDPG with respect to cumulative reward of each episode shown in Fig. 5.2. The values in Fig 5.2 are averaged over 20 experiments for each algorithm. After the comparison between the value of minibatch sizes  $N$  and replay buffer size  $Z$ , 64 and 10000 are chosen to make comparison between DDPG and PE-DDPG respectively. The comparison of minibatch values and replay buffer size values can be seen on the section 2.1.

As can be seen in Fig. 5.2, the proposed PE-DDPG converges earlier on in the learning phase compared to standard DDPG, from earlier episodes such as between 50 and 100. The cumulative reward of first 200 episodes is higher than DDPG algorithm. This result is expected, because, the proposed algorithm prioritizes the samples that our agent is bad at, and therefore, train the agent more with those past experiences where it is performing poorly.

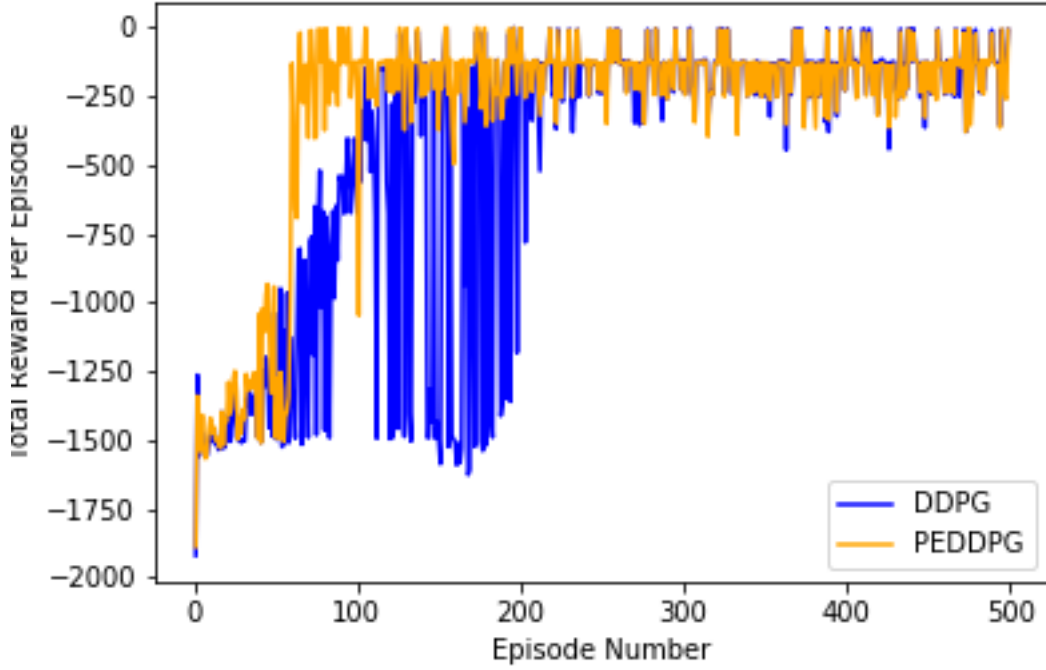


FIGURE 5.2: Total Reward Per Episode of DDPG PE-DDPG

TABLE 5.3: Mean Cumulative Reward of Episode Intervals

Episode Number	50-100	100-150	150-200	200-300	300-400	400-500
DDPG	-768.40	-210.21	-173.23	-156.92	-159.25	-149.05
PE-DDPG	-407.77	-173.67	-174.76	-142.61	-167.86	-151.92

TABLE 5.4: Standard Deviation of Cumulative Reward of Episode Intervals

Episode Number	50-100	100-150	150-200	200-300	300-400	400-500
DDPG	577.40	208.51	105.61	117.70	85.19	90.19
PE-DDPG	500.97	149.06	114.56	76.05	88.30	89.07

The mean cumulative rewards and standard deviations are shown in Tables 5.3 and 5.4 respectively. It can be seen that the mean of cumulative reward and standard deviation of DDPG and PE-DDPG are similar. However, it can also be seen that PE-DDPG attains this performance much earlier in training. This has two benefits: First, in a physical implementation, less number of trials are necessary. Second, if the plant is open loop unstable, a quicker learning allows it to remain more close to the stable region, and therefore collect higher quality samples to learn from; which improves learning speed further. Lower standard deviation earlier on in learning means that

the control is more consistent in PE-DDPG compared to standard DDPG, which is, naturally, more beneficial. However, time of each episode in PE-DDPG is around 2.28 minutes, but, each DDPG’s episode takes 1.8 minutes. The reason to longer episode time is that PE-DDPG’s sampling mechanism needs some computation and increases the time complexity of algorithm.

The TD-errors during training are also of importance. The agent concentrates its learning effort on samples with higher error. Therefore, we also compared the absolute mean of TD-errors of standard DDPG and PE-DDPG. The result can be seen in Table 5.5. Since sampling mechanisms are different, sum of absolute errors of PE-DDPG is lower than DDPG.

TABLE 5.5: Total of Absolute TD-error of Episode Intervals

Episode Number	50-100	100-150	150-200	200-300	300-400	400-500
DDPG	9032.66	5767.79	10036.62	6310.81	4361.93	3369.40
PE-DDPG	4951.06	5059.56	5011.98	2576.96	1879.75	1826.43

TABLE 5.6: Standard Deviation of TD-Error of Episode Intervals

Algorithm/Episode Number	50-100	100-150	150-200	200-300	300-400	400-500
DDPG	1275.61	1327.41	1554.48	2688.77	479.50	486.65
PE-DDPG	1093.74	874.44	1187.26	618.46	372.55	385.67

To conclude, the main reason that the proposed algorithm has higher reward in earlier episodes is that standard DDPG algorithm uses uniform sampling which causes the agent to waste time with samples that are not useful for its existing performance. However, the proposed PE-DDPG algorithm uses those samples with high TD-error, which makes the agent concentrate its training to the areas which it is not good at.

## 5.4 Experiments on Algorithm Parameters

Algorithm parameters, minibatch size  $N$  and replay buffer size  $Z$  are changed during testing of the proposed algorithm. Figure 5.4 shows the importance of the replay buffer size. The values in the graph retrieved with the value of minibatch size as 32.

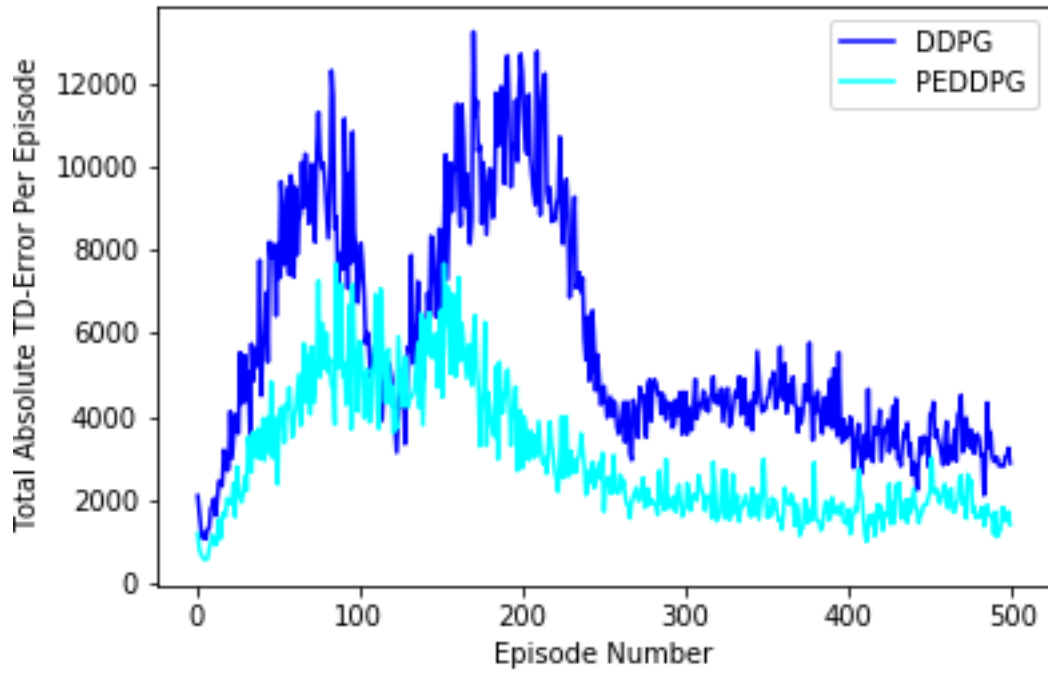


FIGURE 5.3: Comparison of Sum of Absolute TD Error per Episode

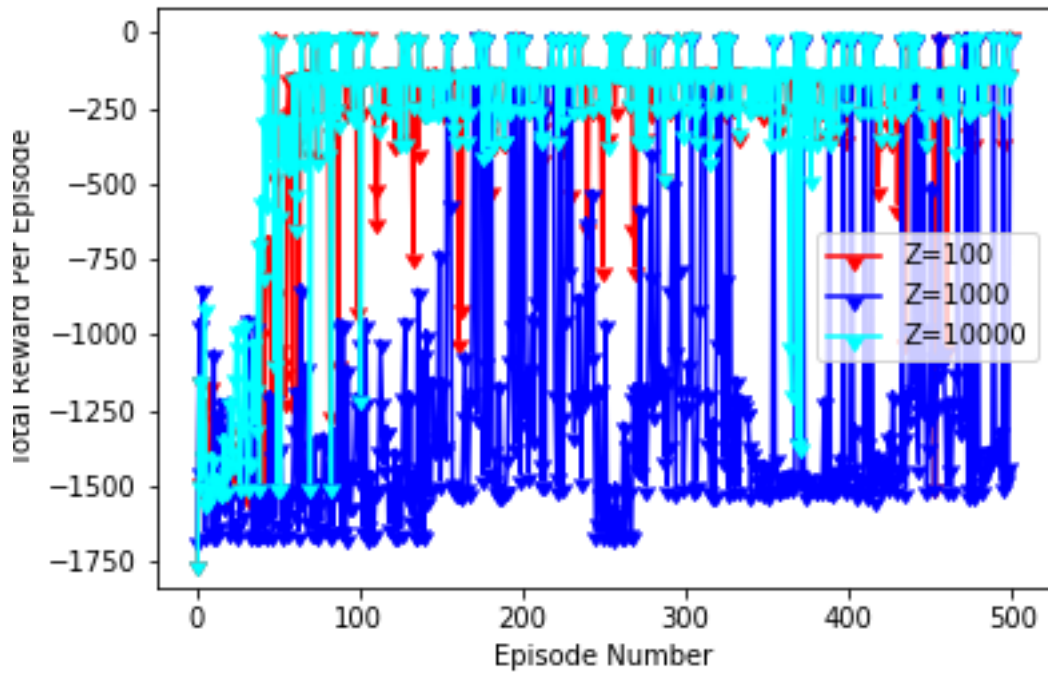


FIGURE 5.4: Comparison of Replay Buffer Size in PE-DDPG



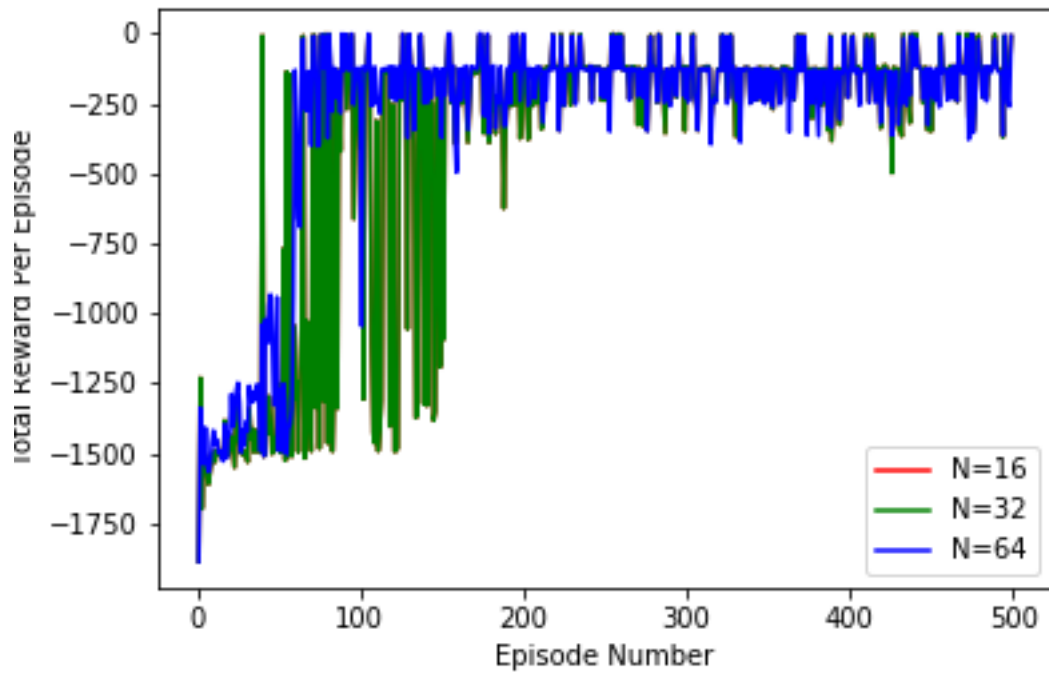


FIGURE 5.5: Comparison of Minibatch Size in PE-DDPG

## Chapter 6

# Conclusions and Future Work

In this thesis, we concentrated on the problem of applying Reinforcement Learning Framework to the control of dynamic plants. We improved the existing Deep Deterministic Policy Gradient method which resamples its past experiences uniformly. An algorithm PE-DDPG which resamples past experiences with higher TD-error more frequently on continuous state space problem. The results show that proposed PE-DDPG has similar performance to the standard DDPG after they both converge. However, the proposed method converges quicker due to importance sampling and the variance of reward is also low, implying better performance during training. This makes the PE-DDPG algorithm more suitable to experimental setups where taking samples from a physical system are costly. It is also beneficial for open-loop unstable plants because earlier convergence of the controller makes it possible to get more meaningful samples from the plant, and further simplifies data collection from physical plants.

As future work, in order to increase efficiency of samples in training of the reinforcement learning agent, PE-DDPG can be improved further: Critic Network. Since Critic Network determines the Q Value of state and action, the network can be thought of as an evaluation function of the policy. Thus, some sample efficient off-policy policy evaluation methods in the literature can be applied to PE-DDPG to make the algorithm more efficient.

# Bibliography

- [1] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *CoRR*, abs/1509.02971, 2015.
- [2] G. Tesauro. Temporal difference learning and td-gammon. *Communications of the ACM*, 38(3):58–68, 1995.
- [3] J. B. Pollack and A. D. Blair. Why did td-gammon work. In *Advances in Neural Information Processing Systems 9*, pages 10–19, 1996.
- [4] C.J.C.H. Watkins and Peter Dayan. Q-learning. *Machine Learning*, 8(3–4):279–292, 1992.
- [5] J. N. Tsitsiklis and B. V. Roy. An analysis of temporal-difference learning with function approximation. *Automatic Control, IEEE Transactions*, 42(5):674–690, 1997.
- [6] Martin Riedmiller. Neural fitted q iteration-first experiences with a data efficient neural reinforcement learning method. *Machine Learning: ECML*, pages 317–328, 2005.
- [7] Long-Ji Lin. Reinforcement learning for robots using neural networks. *Technical report, DTIC Document*, 1993.
- [8] I. Sutskever A. Krizhevsky and G. Hinton. Imagenet classification with deep convolutional neural networks. *Neural Information Processing Systems*, 25:1106–1114, 2012.
- [9] S. Chintula P. Sermanet, K. Kavukcuoglu and Y. LeCun. Pedestrian detection with unsupervised multi-stage feature learning. *International Conference on Computer Vision and Pattern Recognition*, 2013.
- [10] H. Kita A. Onat and Y. Nishikawa. Q-learning with recurrent neural networks as a controller for the inverted pendulum problem. *The Fifth International Conference on Neural Information Processing*, pages 837–840, 1998.

- [11] Volodymyr Mnih. Machine learning for aerial image labeling. *Phd thesis, University of Toronto*, pages 837–840, 2013.
- [12] D. Silver A. Graves I. Antonoglou D. Wierstra V. Mnih, K. Kavukcuoglu and M. Riedmiller. Playing atari with deep reinforcement learning. *NIPS Deep Learning Workshop*, 2013.
- [13] D. Silver A. A. Rusu J. Veness M. G. Bellemare A. Graves M. Riedmiller A. K. Fidjeland G. Ostrovski S. Petersen C. Beattie A. Sadik I. Antonoglou H. King D. Kumaran D. Wierstra S. Legg V. Mnih, K. Kavukcuoglu and D. Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.
- [14] A. Guez H. van Hasselt and D. Silver. Deep reinforcement learning with double q-learning. *Association for the Advancement in Artificial Intelligence*, 2015.
- [15] D. Silver. Lecture notes on reinforcement learning, 2015.
- [16] Jan Peters. Policy gradients methods. *Scholarpedia*, 5(11), 2010.
- [17] J.C. Spall. Introduction to stochastic search and optimization: Estimation, simulation, and control. *Hoboken, NJ: Wiley*, 2003.
- [18] T.Rckstiess A. Graves J.Peters F.Seinke, C.Osendorfer and J. Schmidhuber. Parameter-exploring policy gradients. *Neural Networks*, 4(23):551–559, 5 2010.
- [19] R.J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, pages 229–256, 1992.
- [20] J.Peters and S.Schaal. Reinforcement learning of motor skills with policy gradients. *Neural Networks*, 4(21):682–697, 2008.
- [21] V.R. Konda and J.N. Tsitsiklis. Actor-critic algorithms. *SIAM Journal on Control and Optimization*, 42(21):1143–1166, 2003.
- [22] G A. Rummery and Mahesan Niranjana. On-line q-learning using connectionist systems. *Technical Report CUED/F-INFENG/TR 166*, 11 1994.
- [23] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. *CoRR*, abs/1602.01783, 2016. URL <http://arxiv.org/abs/1602.01783>.
- [24] John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation. *ICPR*, 06 2015.

- [25] Richard S. Sutton and Andrew G. Barto. *Introduction to Reinforcement Learning*. MIT Press, Cambridge, MA, USA, 1st edition, 1998. ISBN 0262193981.
- [26] David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. Deterministic policy gradient algorithms. In Eric P. Xing and Tony Jebara, editors, *Proceedings of the 31st International Conference on Machine Learning*, volume 32/1 of *Proceedings of Machine Learning Research*, pages 387–395, Beijing, China, 22–24 Jun 2014. PMLR. URL <http://proceedings.mlr.press/v32/silver14.html>.
- [27] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. *CoRR*, abs/1511.05952, 2016.
- [28] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D. Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, Xin Zhang, Jake Zhao, and Karol Zieba. End to end learning for self-driving cars. *CoRR*, abs/1604.07316, 2016. URL <http://arxiv.org/abs/1604.07316>.
- [29] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. In *International Conference on Learning Representations*, 2015.
- [30] Geoffrey Hinton, Li Deng, Dong Yu, George Dahl, Abdel rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara Sainath, and Brian Kingsbury. Deep neural networks for acoustic modeling in speech recognition. *Signal Processing Magazine*, 2012.
- [31] Shuai Zhang, Lina Yao, Aixin Sun, and Yi Tay. Deep learning based recommender system: A survey and new perspectives. *ACM Comput. Surv.*, 52(1): 5:1–5:38, February 2019. ISSN 0360-0300. doi: 10.1145/3285029. URL <http://doi.acm.org/10.1145/3285029>.
- [32] Dean Pomerleau, Jay Gowdy, and Charles E. Thorpe. Combining artificial neural networks and symbolic processing for autonomous robot guidance. *Engineering Applications of Artificial Intelligence*, 4:279–285, 12 1991. doi: 10.1016/0952-1976(91)90042-5.
- [33] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2014. URL <http://arxiv.org/abs/1412.6980>. cite arxiv:1412.6980Comment: Published as a conference paper at the 3rd International Conference for Learning Representations, San Diego, 2015.
- [34] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for on-line learning and stochastic optimization. *J. Mach. Learn. Res.*, 12:2121–2159, July

2011. ISSN 1532-4435. URL <http://dl.acm.org/citation.cfm?id=1953048.2021068>.
- [35] Xavier Glorot, Antoine Bordes, and Y Bengio. Deep sparse rectifier neural networks. *Journal of Machine Learning Research*, 15, 01 2010.

